There are 3 approaches through which Entity framework is implemented.

1. Database First
2. Code First
3. Model First

Of these Database first and Code First  are the most used ones. In this article I will not discuss model first approach.

First let us understand what are code first and database first

**Code First Approach**In code first approach we will first create entity classes with properties defined in it. Entity framework will create the database and tables based on the entity classes defined. So database is generated from the code. When the dot net code is run database will get created.

**Advantages**

1. You can create  the database and tables from your business objects.
2. You can specify which related collections are to be eager loaded, or not be serialized at all.
3. Database version control.
4. Good for small applications.

**Disadvantages**

1. You have to write everything related to database in the visual studio code.
2. For stored procedures you have to map stored procedure using Fluent API and write Stored Procedure inside the code.
3. If you want to change anything in the database tables you to make changes in the entity classes in the code file and run the update-database from the  package manager console.
4. Not preferred for Data intensive applications.

**Database First Approach**In this approach Database and tables are created first. Then you create entity Data Model using the created database.

**Advantages**

1. Simple to create the data model
2. Graphical  user interface.
3. Mapping and creation of keys and relationships are easy as you need not have to write any code .
4. Preferred for data intense and large applications

**Disadvantages**

1. Using an existing database to generate a .edmx model file and the associated code models results in a giant pile of auto generated code.
2. When you need to add any functionality to generated model you have to extend the model class generated.

Choosing the appropriate approach is purely based on the applications you are developing.

**Entity Framework 6**

Entity Framework 6 (EF6) is a tried and tested data access technology. It was first released in 2008(3.5 , 4.0,5.0,6.0) , as part of .NET Framework 3.5 SP1 and Visual Studio 2008 SP1. Starting with the 4.1 release it has shipped as the [EntityFramework](https://www.nuget.org/packages/EntityFramework/) NuGet package. EF6 runs on the .NET Framework 4.x, which means it runs only on Windows.

EF6 continues to be a supported product, and will continue to see bug fixes and minor improvements.

**Entity Framework Core**

Entity Framework Core (EF Core) is a complete rewrite of EF6 that was first released in 2016. It ships in Nuget packages, the main one being [Microsoft.EntityFrameworkCore](https://www.nuget.org/packages/Microsoft.EntityFrameworkCore/). EF Core is a cross-platform product that can run on .NET Core or .NET Framework.

EF Core was designed to provide a developer experience similar to EF6. Most of the top-level APIs remain the same, so EF Core will feel familiar to developers who have used EF6.

--------------------------------------------------------------------------------------------

## Introduction

We all know Databases is heart of any application, and some time its too complex to manage right? Yes. That’s why Microsoft come up with the **Entity Framework** :  which reduce complexity and make easy for developers to write great program without having much more idea of Database. Thats why now-a-days most of Microsoft .Net developers use the .Net Entity Framework for database operation.

### ****What is the Entity framework?****

The Entity framework is an **object-relation mapper**, means - it takes the structure of the database and turns it into objects that the .Net framework can understand. Developers use those object to interact with the database instead of interacting with database directly. It’s possible to perform the full set of Create, Read, Update, and Delete (CRUD) operations using the Entity Framework features.

It has three work flows

1. **Code – first,**
2. **Model-first and**
3. **database first.**

Choosing and using the right approach can save developers a lot of time and effort, especially when interacting with complex database design.

In this Article we will work through the **Code-first work flow** with simple Demo Application.

**Cover part:**

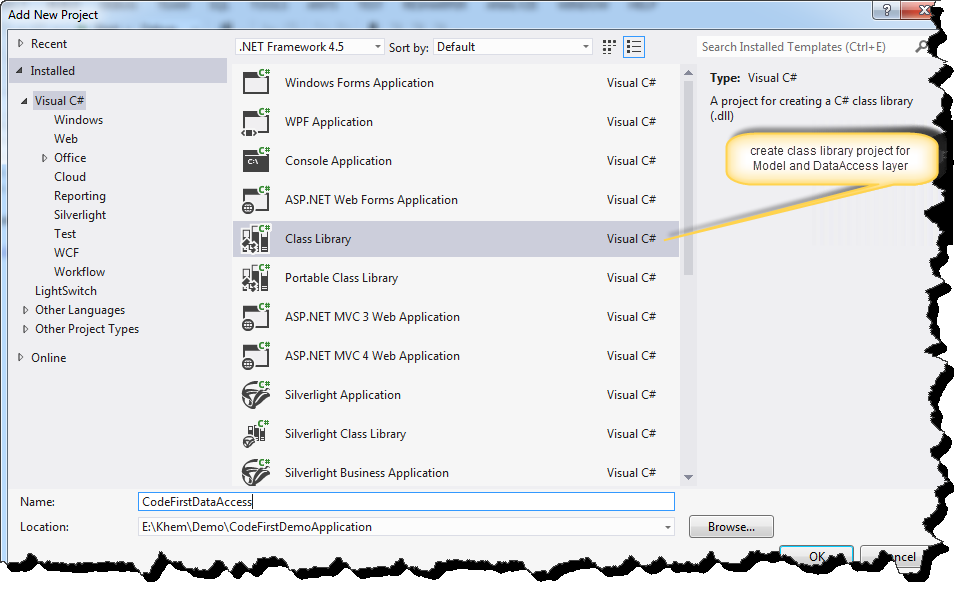
1. Understanding the Code First Work-flow
2. Simple Demonstration on code-frist approach with MVC Application.

## ****Understanding the Code First Work-flow****

The code first approach is introduced in **Entity Framework 4.1**, and is a latest workflow Microsoft has introduced. It lets us transform our coded classes into a database application, which means code first lets us to define our domain model using POCO (plan old CLR object) class rather than using an xml - based EDMX files, which has no dependency with entity framework. Our model classes becomes the domain model, there for we most have to be very conscious in designing our model classes. And the rest work will be done by entity-frame work. This is the beauty of the code first approach where our model classes are become the data models on which Entity framework relies.

**Simple Demonstration on code-frist approach with Asp.Net MVC Application**

1) Create a class library project Named as “**CodeFirstModel**” and solution Named as “**CodeFirstDemoApplication**”. Open VS -> File ->New Project -> Class library project. Refer screen shot below.



And new class named as “Employee.cs”. Project name->right click-> Add-> class.

Hide   Copy Code

publicclass Employee

{

    public int EmployeeId { get; set; }

    public string Name { get; set; }

    public string Description { get; set; }

    public bool IsEmployeeRetired { get; set; }

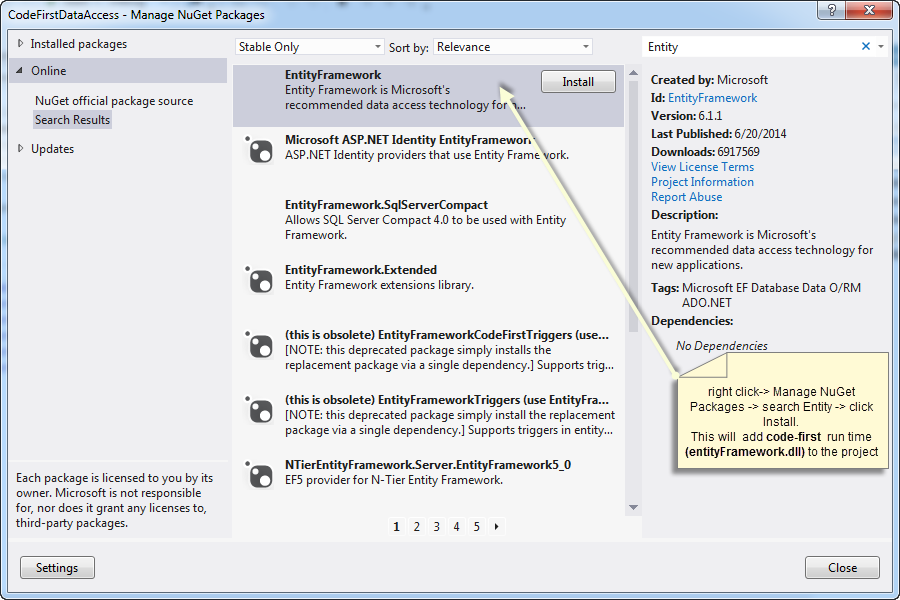
    public string Country { get; set; }

    public string Company { get; set; }

}

Code-first will use this model class to define data model on which entity framework relies. It has convention (we will discuss more in next article) that if it finds a property named **Id**or a property with the combined name of **type name and Id**  (i.eEmployeeId) that property will be automatically configured as the Primary key and not null and also this key marked as auto-increment identity in the table. If it cannot find any property that matches this convention, it will throw an exception at run-time telling there is no key (I.eEntityType “class name” has no key defied. Define the key for this entity type).

2) Create another Class library project and named as “**CodeFirstDataAccess**” : Right click project solution ->  Add -> new Project -> Class library project and **Install Entity framework**form NuGet Package manager. Select Project -> right click -> Manage NuGet Packages -> Search Entity FrameWork -> Install. refer below screen shot.



i) Add the reference of **entity framework dll** and “**CodeFirstModel**” to the current project "**i.eCodeFirstDataAccess**".  
ii) Add a class named as “**DemoEntityContext.cs**” - Then implement **DBContext**and **DBSet**to interact with database and its operations.

Hide   Copy Code

publicclass DemoEntityContext : DbContext

{

    public DemoEntityContext() : base("name=DbConnectionString") {    }

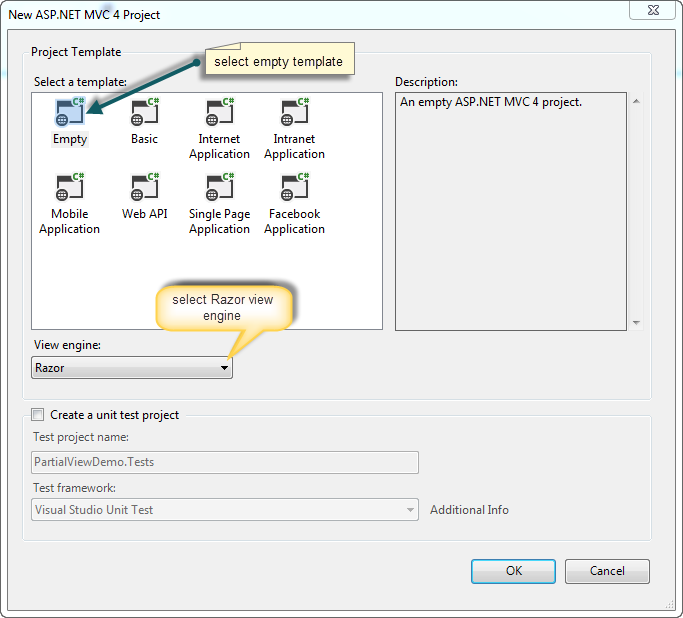
    public DbSet<Employee> Employees { get; set; }

}

This simple class represents the whole data layer for  our application. ”**DBConnectionString**” is the connection string name defined in web.configfile(discuss later).

**DBContext**– In simple this class is responsible to interact with database, and also to manage the entity objects during run time, which includes populating objects with data from a database, change tracking, and persisting data to the database.  
  
**DBSet**- This class represents an entity set that is used for the create, read, update, and delete operations.

3) Add an Empty MVC web application named as “CodeFirstWebApp” which will use the above datalayer for db operation. Right click on project solutions -> Add -> new Project -> ASP.NET MVC4 Web application. refer screen shot



Enable-migration

Add-migration filename –context contextname

Update-database

# Configure One-to-Many Relationships in EF 6

Here, we will learn how to configure One-to-Many relationships between two entities (domain classes) in Entity Framework 6.x using the code-first approach.

Let's configure a one-to-many relationship between the following Student and Grade entities where there can be many students in one grade.

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

}

public class Grade

{

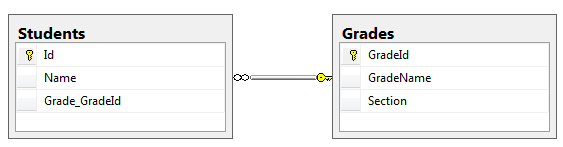
public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

}

After implementing the one-to-many relationship in the above entities, the database tables for Student and Grade will look like below.

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-db.PNG)

The one-to-many relationship can be configured in the following ways.

1. [By following Conventions](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#conventions-for-one-to-many-ef6)
2. [By using Fluent API Configurations](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#configure-one-to-many-using-fluent-api)

## Conventions for One-to-Many Relationships

There are certain conventions in Entity Framework which if followed in entity classes (domain classes) will automatically result in a one-to-many relationship between two tables in the database. You don't need to configure anything else.

Let's look at an example of all the conventions which create a one-to-many relationship.

### Convention 1

We want to establish a one-to-many relationship between the Student and Grade entities where many students are associated with one Grade. It means that each Student entity points to a Grade. This can be achieved by including a reference navigation property of type Grade in the Student entity class, as shown below.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

**public Grade Grade { get; set; }**

}

public class Grade

{

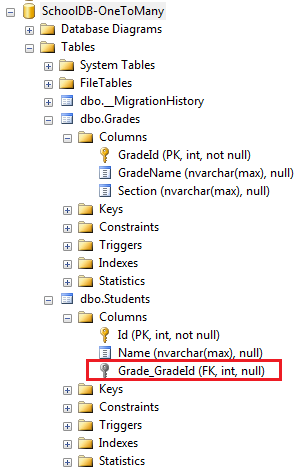
public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

}

In the above example, the Student class includes a reference navigation property of Grade class. So, there can be many students in a single grade. This will result in a one-to-many relationship between the Students and Grades table in the database, where the Students table includes foreign key Grade\_GradeId as shown below.

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-1.PNG)

Notice that the reference property is nullable, so it creates a nullable foreign key column Grade\_GradeId in the Students table. You can [configure NotNull foreign key using fluent API](https://www.entityframeworktutorial.net/code-first/configure-one-to-many-relationship-in-code-first.aspx#notnull-foreignkey-using-fluent-api).

### Convention 2

Another convention is to include a collection navigation property in the principal entity as shown below.

public class Student

{

public int StudentId { get; set; }

public string StudentName { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

**public ICollection<Student> Students { get; set; }**

}

In the above example, the Grade entity includes a collection navigation property of type ICollection<Student>. This also results in a one-to-many relationship between the Student and Grade entities. This example produces the same result in the database as convention 1.

### Convention 3

Including navigation properties at both ends will also result in a one-to-many relationship, as shown below.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

**public Grade Grade { get; set; }**

}

public class Grade

{

public int GradeID { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

**public ICollection<Student> Student { get; set; }**

}

In the above example, the Student entity includes a reference navigation property of the Grade type and the Grade entity class includes a collection navigation property of the ICollection<Student> type which results in a one-to-many relationship. This example produces the same result in the database as convention 1.

### Convention 4

A fully defined relationship at both ends will create a one-to-many relationship, as shown below.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

**public int GradeId { get; set; }**

**public Grade Grade { get; set; }**

}

public class Grade

{

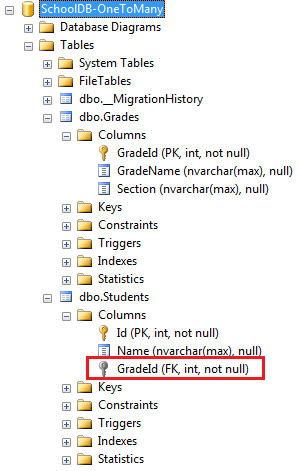
public int GradeId { get; set; }

public string GradeName { get; set; }

**public ICollection<Student> Student { get; set; }**

}

In the above example, the Student entity includes foreign key property GradeId with its reference property Grade. This will create a one-to-many relationship with the NotNull foreign key column in the Students table, as shown below.

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-2.PNG)

If the data type of GradeId is nullable integer, then it will create a null foreign key.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public int? GradeId { get; set; }

public Grade Grade { get; set; }

}

The above code snippet will create a nullable GradeId column in the database because we have used Nullable<int> type (? is a shortcut for Nullable<int>)

## Configure a One-to-Many Relationship using Fluent API

Generally, you don't need to configure the one-to-many relationship in entity framework because one-to-many relationship conventions cover all combinations. However, you may configure relationships using Fluent API at one place to make it more maintainable.

Consider the following Student and Grade entity classes.

public class Student

{

public int Id { get; set; }

public string Name { get; set; }

public int CurrentGradeId { get; set; }

public Grade CurrentGrade { get; set; }

}

public class Grade

{

public int GradeId { get; set; }

public string GradeName { get; set; }

public string Section { get; set; }

public ICollection<Student> Students { get; set; }

}

You can configure a one-to-many relationship for the above entities using Fluent API by overriding the OnModelCreating method in the context class, as shown below.

public class SchoolContext : DbContext

{

public DbSet<Student> Students { get; set; }

public DbSet<Grade> Grades { get; set; }

protected override void OnModelCreating(DbModelBuilder modelBuilder)

{

// configures one-to-many relationship

modelBuilder.Entity<Student>()

.HasRequired<Grade>(s => s.CurrentGrade)

.WithMany(g => g.Students)

.HasForeignKey<int>(s => s.CurrentGradeId); }

}

}

Let's understand the above code step by step.

* First, we need to start configuring with any one entity class. So, modelBuilder.Entity<student>() starts with the Student entity.
* Then, .HasRequired<Grade>(s => s.CurrentGrade) specifies that the Student entity has required the CurrentGrade property. This will create a NotNull foreign key column in the DB.
* Now, it's time to configure the other end of the relationship - the Grade entity.
* .WithMany(g => g.Students) specifies that the Grade entity class includes many Student entities. Here, many infers the ICollection type property.
* Now, if the Student entity does not follow the Id property convention for foreign key, then we can specify the name of the foreign key using the HasForeignKey method.
* .HasForeignKey<int>(s => s.CurrentGradeId); specifies the foreign key property in the Student entity.

Alternatively, you can start configuring the relationship with the Grade entity instead of the Student entity. The following code produces the same result as above.

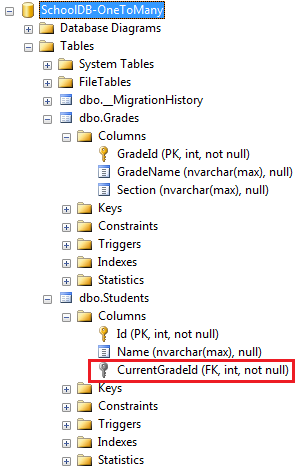
modelBuilder.Entity<Grade>()

.HasMany<Student>(g => g.Students)

.WithRequired(s => s.CurrentGrade)

.HasForeignKey<int>(s => s.CurrentGradeId);

The above example will create the following tables in the database.

[](https://www.entityframeworktutorial.net/images/codefirst/onetomany-4.PNG)

### Configure the NotNull ForeignKey using Fluent API

In convention 1, we have seen that it creates an optional one-to-many relationship which in turn creates a nullable foreign key column in the database. To make it a NotNull column, use the HasRequired() method as shown below.

modelBuilder.Entity<Student>()

.HasRequired<Grade>(s => s.CurrentGrade)

.WithMany(g => g.Students);

### Configure Cascade Delete using Fluent API

Cascade delete means automatically deleting child rows when the related parent row is deleted. For example, if Grade is deleted then all the students in that Grade should also be deleted automatically. The following code configures the cascade delete using the WillCascadeOnDelete method.

modelBuilder.Entity<Grade>()

.HasMany<Student>(g => g.Students)

.WithRequired(s => s.CurrentGrade)

.WillCascadeOnDelete();